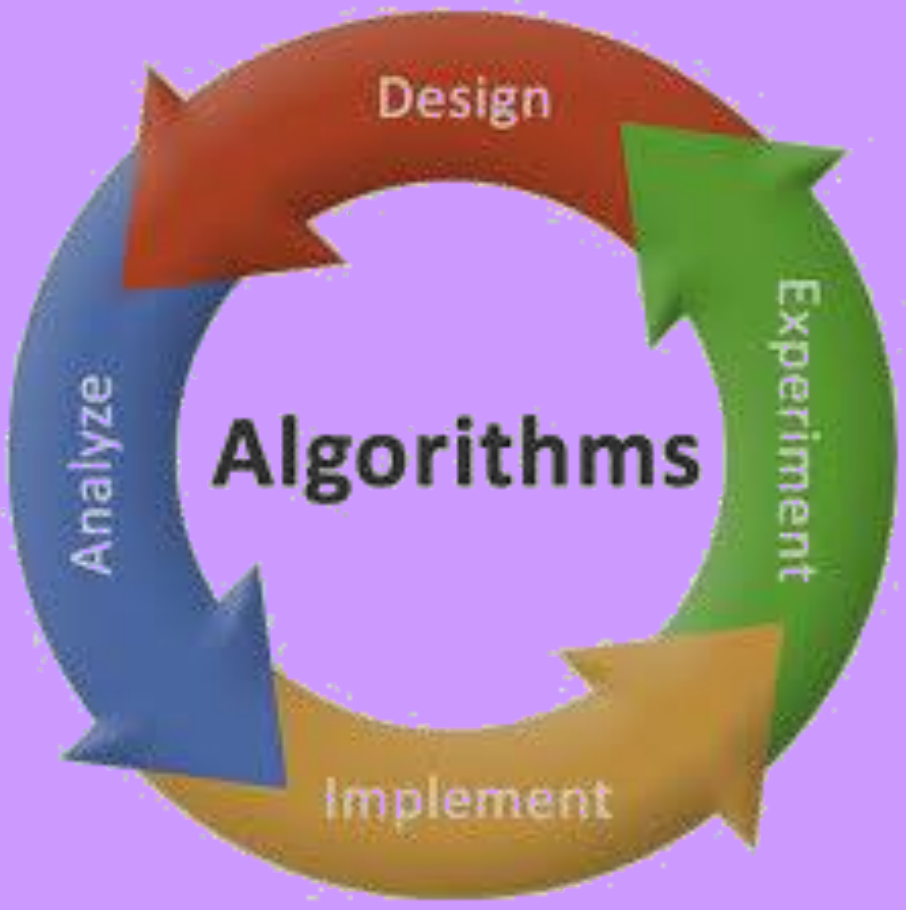
THE MAGICAL GUIDE TO ALGORITHM ANALYSIS AND DESIGN



Rosina S Khan

Dedicated to You,

The Valued Reader

Copyright Information

https://rosinaskhan.weebly.com

Table of Contents

Preface

I have named this guide as "The Magical Guide to Algorithm Analysis and Design" because the very process of writing down a set of instructions in the form of pseudocodes before feeding them into program structures and executing them successfully to get program outputs is very magical indeed. It all starts with the human thinking process and via pseudocodes or algorithms, converting them into programs, including their analysis and design, is nothing short of magic and therefore, the title of this guide.

Analysis is the measurement of the quality of your design. Just like you use your sense of taste to check your cooking, you should get into the habit of using algorithm analysis to justify design decisions when you write an algorithm or a computer program. This is a necessary step to reach the next level in mastering the art of programming.

I have integrated the content of this book from various resources including several google searches. The main titles I have used for this guide are the following:

1) Lecture Notes for Data Structures and Algorithms by John Bullinaria

2) A Handout on Introduction to Data Structures and Algorithms by IDOL

3) Lecture Notes on Algorithm Analysis and Design by Herbert Edelsbrunner

4) A Handout on Introduction to Algorithms by Jon Kleinberg and Eva Tardos

5) A Common-Sense Guide to Data Structures and Algorithms by Jay Wengrow, 2nd Edition, The Pragmatic Programmers

The guide, authored by me, is meant for undergraduate students in the field of Computer Science and Engineering or an equivalent program.

This book is organized into 11 chapters.

In Chapter 1, we introduce the concept of Algorithms and the fundamental questions about algorithms. [1]

Chapter 2 portrays the concept of Data Structures and their varieties such as arrays, linked lists, stacks, and queues. Also, abstract data type and the advantages of abstract data trees are explained. [2, 5]

In Chapter 3, we formally define an algorithm, introduce time and space complexities, the types of analyses of algorithms, and mathematical notations. [2, 5]

Chapter 4 depicts searching algorithms such as linear search and binary search. [1]

In Chapter 5, we discuss the concept of trees both quadtrees and binary trees. [1]

Chapter 6 focuses on binary search trees- how to build and search them, how to sort using them, how to delete nodes from them, and we introduce B-trees. [1]

Chapter 7 deals with the various operations on binary heap trees. [1]

The next chapter, Chapter 8 concentrates on the various sorting techniques such as, Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort, and Heap sort and their algorithmic complexities as well. [2, 5]

Chapter 9 goes on to explain graphs: the basic concepts, terminology used, representations, operations (Depth-First Search and Breadth First Search) and traversals. [2]

Chapter 10 talks about the concepts of spanning tree and minimum spanning tree and their applications, and also, takes into account graph algorithms such as Kruskal's algorithm and Prim's algorithm based on the above concepts. [2, 5]

Chapter 11 is concerned with the theoretical aspects of various Algorithm Design Techniques such as, Divide and Conquer, Backtracking, Dynamic Programming and Greedy Methods. [2, 3, 4, 5]

At the end of the guide, I cater to further free resources, which you will find both valuable and enjoyable.

Acknowledgements

I am extremely grateful to John Bullinaria, IDOL, Herbert Edelsbrunner, Jon Kleinberg and Eva Tardos, and Jay Wengrow for using some of their resources and merging with what I have and therefore, the creation of this guide.

Last but not the least, I am thankful to my family for their support during the write-up of this eBook.

C H A P T E R 1

Introduction

1. Introduction to Algorithms

The following chapters cover the key ideas involved in designing algorithms. We shall see how they depend on the design of suitable data structures, and how some structures and algorithms are more efficient than others for the same task. We will concentrate on a few basic tasks, such as storing, sorting, and searching data, that underlie much of computer science, but the techniques discussed will be applicable much more generally.

Throughout, we will investigate the computational efficiency of the algorithms we develop, and gain intuitions about the pros and cons of the various potential approaches for each task. We will not restrict ourselves to implementing the various data structures and algorithms in particular computer programming languages (e.g., Java, C etc.), but specify them in simple pseudocode that can easily be implemented in any appropriate language.

1.1 Algorithms as Opposed to Programs

An algorithm for a particular task can be defined as a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

As such, an algorithm must be precise enough to be understood by human beings. However, in order to be executed by a computer, we will generally need a program that is written in a rigorous formal language; and since computers are quite inflexible compared to the human mind, programs usually need to contain more details than algorithms.

Here, we shall ignore most of those programming details and concentrate on the design of algorithm rather than programs. The task of implementing the discussed algorithms as computer programs is important, of course, but these chapters will concentrate on the theoretical aspects and leave the practical programming aspects to be studied elsewhere.

Having said that, we will often find it useful to write down segments or whole of actual programs in some areas in order to clarify and test certain theoretical aspects of algorithms and their data structures.

Algorithms can obviously be described in plain English, and we will sometimes do that. However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted. This is called pseudocode, which comes in a variety of forms. Often the chapters will present segments of pseudocode that are very similar to the languages we are mainly interested in, namely the overlap of C and Java, with the advantage that they can easily be inserted into runnable programs.

1.2 Fundamental Questions About Algorithms

Given an algorithm to solve a particular problem, we are naturally led to ask:
1. What is it supposed to do?
2. Does it really do what it is supposed to do?
3. How efficiently does it do it?

The technical terms normally used for these three aspects are:
1. Specification.
2. Verification.
3. Performance analysis.

The details of these three aspects will usually be rather problem dependent.

The specification should formalize the crucial details of the problem that the algorithm is intended to solve. Sometimes that will be based on a particular representation of the associated data, and sometimes it will be presented more abstractly. Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous.

For simple problems, it is often easy to see that a particular algorithm will always work i.e., that it satisfies its specification. However, for more complicated specifications and/or algorithms, the fact that an algorithm satisfies its specification may not be obvious at all. In this case, we need to spend some effort verifying whether the algorithm is indeed correct. In general, testing on a few particular inputs can be enough to show that the algorithm is incorrect.

Finally, the efficiency or performance of an algorithm relates to the resources required by it, such as how quickly it will run, or how much computer memory it will use. This will usually depend on the problem instance size, the choice of data representation, and the details of the algorithm. Indeed, this is what normally drives the development of new data structures and algorithms. We shall study the general ideas concerning efficiency and then apply them throughout the remainder of the chapters.

C H A P T E R 2

Data Structures

2. Introduction

What are data structures? They are structured data with logical relationships between data elements. For example, a street address can be identified by street number and street name. These structured data variables depend on one another to create a unique structure, the street address. In data structure, a linked list, for example, would link data elements to form a structured component of the system.

2.1 Basic Terminology

We use some terminology while working with data structures, which should be obvious to every Computer Science undergraduate student. These are explained below:

Data: Data can be defined as a fundamental value, or a collection of values. For example, data about a student can be his student id and name.

Group/Composite Item: A group or composite item has parts or subordinate items. For example, a student's name can have first name, middle name, and last name as parts or subordinate items so that the composite/group item here is the student's name.

Attribute/Entity: An entity has properties or attributes. For example, the student entity has properties or attributes as name, id, address, phone number etc.

Field/Record: The student entity actually becomes a collection of records in a file. For each of them pertaining to a specific student. The fields of a record are the data or attributes of a student for example, id, name, address, phone number.

2.2 The Need for Data Structure

We need a data structure for an organization because:

1) It helps to define the different levels of the organization.
2) It provides a means of storing the data, and also retrieving them at the elementary core.
3) It helps to carry out operations on the stored data such as, deleting, updating, or adding items, or even extracting the highest/lowest priority data item.
4) It helps to store huge amounts of data efficiently.
5) It enables searching and sorting of data conveniently.

2.3 The Goals of Data Structure

1) Whatever problems the organization needs to solve, the data structure does so correctly for all kinds of input.
2) The data structure needs to be efficient as well. It must process the data at a high speed without utilizing much of memory space.
3) Implementation of the data structure may require a certain amount of programming effort.

2.4 Steps in Selecting a Data Structure

1) Analyze your problem to support the basic operations.
2) Quantify the data constraints for the operations involved.
3) Select the data structure that best satisfies the requirements.

The first concern is the data and the data operations. The next concern is the representation of those data, and the final concern is the implementation of the representation.

2.5 Classification of Data Structures

A data structure represents the relationship between data elements and helps programmers to process data easily.

There are mainly two types of data structures:

1) Primitive Data Structures
2) Non-primitive Data Structures

Fig 2.1 shows the different classifications of data structures.

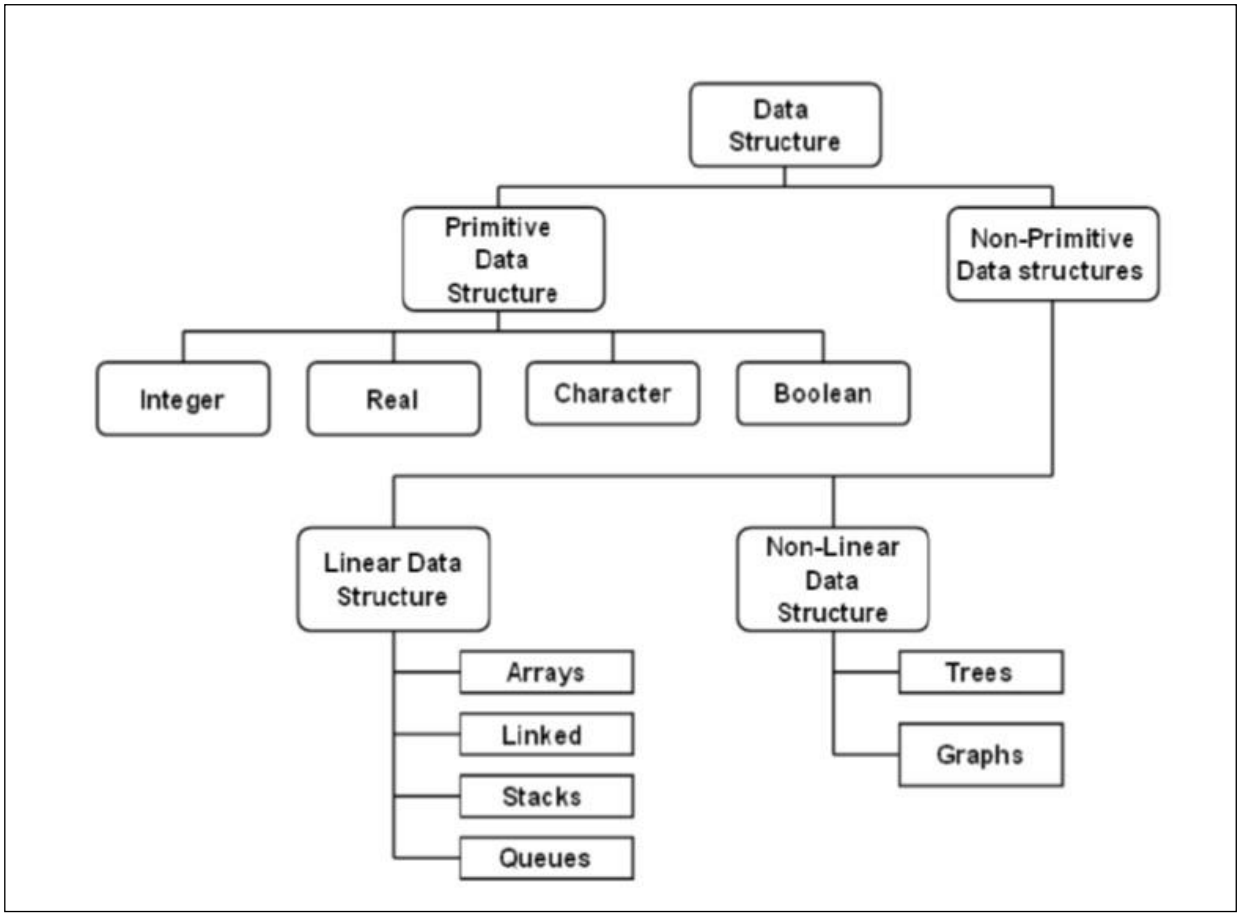


Fig 2.1: Classification of Data Structures

2.6 Primitive Data Structure

Basic data types such as integer, real, character and Boolean fall under primitive data structure category. These data types are simple because they all consist of characters and cannot be further divided. They can be manipulated and operated by machine level instructions.

2.7 Non-Primitive Data Structures

Non-primitive data structures are derived from primitive ones. They are based on data elements that can of the same data type (homogeneous) or different data types (heterogeneous). They cannot be operated by machine level instructions. They can be further divided into linear and non-linear data structures, depending on the structure and arrangement of data.

2.7.1 Linear Data Structures

A data structure that maintains a linear relationship among its data elements is a linear data structure. However, in memory, the data may not be sequential. Examples of these can be array, linked list, stack, queue.

2.7.2 Non-Linear Data Structures

This data structure does not consist of data elements in a linear fashion. Rather they are arranged in a hierarchical arrangement. Insertion and deletion of data items cannot be done here in a linear way. Examples of non-linear data structures are trees and graphs. They will be explained in detail in later chapters.

Array

An array is an orderly arrangement of data elements. These data elements are stored in adjacent locations of the data structure. They are stored linearly with the same data type. So, an array is also called a linear homogeneous data structure.

We can declare an array Arr with 6 values as follows:

int Arr[6]= {56, 17, 60, 9, 7, 10}

This declaration will create an array as shown in the following figure:

Arr	0	1	2	3	4	5
	56	17	60	9	7	10

Fig. 2.2: An Array

Arrays can be classified as one-dimensional, two-dimensional, or multidimensional.

- **One-dimensional Array:** It has only one row of elements. It is stored in ascending storage locations.
- **Two-dimensional Array:** It consists of multiple rows and columns of data elements. It is also called as a matrix.
- **Multidimensional Array:** Multidimensional arrays can be defined as an array of arrays. Multidimensional arrays are not bounded to two indices or two dimensions.
  They can include as many indices as required.

Limitations of Arrays

1) Arrays are of fixed size.
2) Data elements are stored in contiguous memory locations, but they may not always be available.
3) Insertion and deletion of data elements may be problematic because they need to be shifted from their locations.

These limitations can, however, be solved by the use of linked lists.

Applications

1) Storing data elements of the same data type.
2) Auxiliary storage for other data structures.
3) Storage of binary elements of fixed count.
4) Storage of matrices.

Linked List

A linked list is a data structure in which a data element points or links to the next data element of the list. Here data elements need not have consecutive memory locations. Insertion and deletion of data elements are possible anywhere in the linked list. It allocates a block of memory for each data item. For this reason, a linked list is considered a chain of data elements or records called nodes. Each node contains information and pointer fields. The information field contains the actual data while the pointer field contains a pointer to the next node.

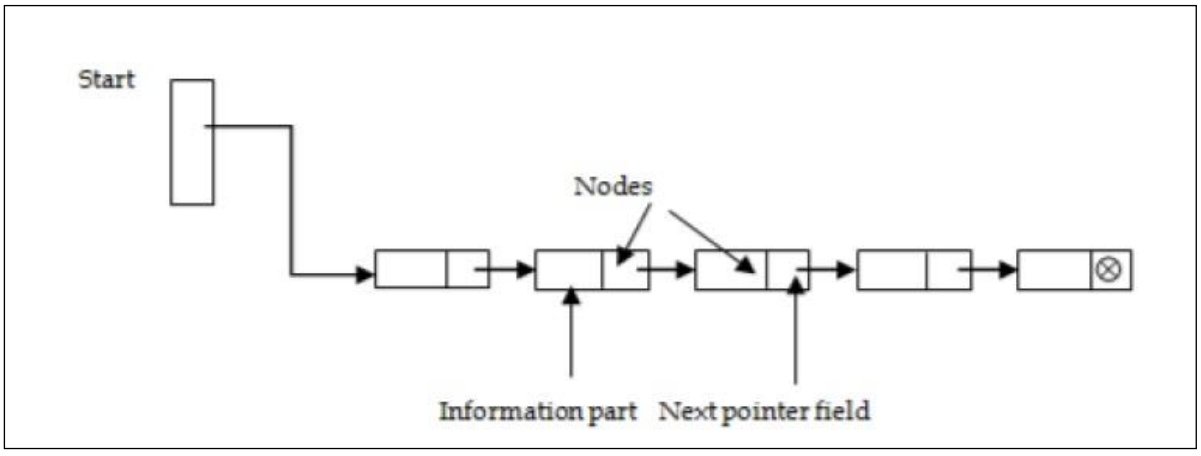


Fig. 2.3: A Linked List

Advantage: Easy to insert and delete data items.

Disadvantage: Searching a data item requires extra memory space and is slow.

Applications

1) Implement stacks, queues, binary trees, and graphs of predefined size.
2) Implement dynamic memory management functions of the operating system (OS).
3) Circular linked list can implement OS or application functions for round robin execution of tasks.
4) Doubly linked list is used in the implementation of forward and backward buttons of a browser.

Stack

A stack is a linear data structure in which insertion and deletion occur at the top of the stack. It is called a last-in first-out (LIFO) data structure because the last element that is pushed on to the top of the stack is always the first one to be popped off or deleted.

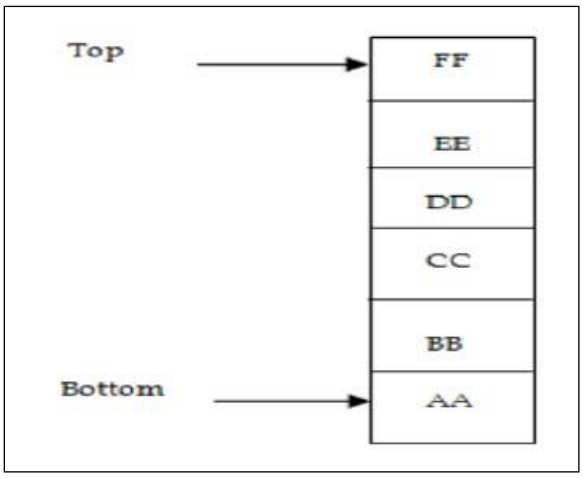


Fig. 2.4: A Stack

In the computer's memory, stacks can be implemented using arrays or linked lists. Fig. 2.4 shows the schematic diagram of a stack. FF is the top of the stack and AA is the bottom of the stack. Since the stack is implemented in a LIFO pattern, data element EE cannot be popped off or deleted before FF. Similarly, DD cannot be deleted before EE.